

Автоматизированный поиск багов в C/C++

Как экономить на своих ногах

О чём пойдёт речь

- Описание проблем и откуда они растут
- Способы решения этих самых проблем (и проблемы от этих решений)
- Опыт применения этих средств из жизни (если успею)
- Как это может быть вам полезно

Проблемы и откуда растут

- Языки C/C++ сложные (особенно C++)
- Из-за этого программисты ошибаются
- Ошибки не всегда находятся компилятором
- Опасность везде - undefined/unspecified behavior
- А должное качество ПО надо как-то обеспечивать
- Тестами всё покрыть очень сложно/долго/затратно/невозможно
 - И можно даже сделать ошибку в тестах :)



HOW TO CODE WITH NO BUGS



Типичные отстрелы ног

- Разыменование нулевого указателя
- Использование невалидного указателя/итератора
- Использование значения переменной после перемещения
- Переполнение знаковых чисел
- Нарушение контрактов стандартных алгоритмов
- Многократное освобождение памяти
- И много чего ещё :)



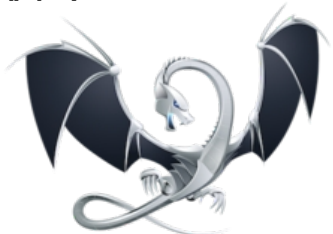
Почему меня это должно волновать

- Ядра ОС
- Браузеры
- Программы для работы с графикой\звуком
- Плееры
- Утилиты для разработки (IDE, компиляторы)
- Мне лень перечислять дальше

Это всё **зачастую** написано с использованием C/C++

Компиляторы

- Используйте больше компиляторов (Clang/GCC/etc)
- Включите больше предупреждений в ваших компиляторах
 - Будьте аккуратны с внешними зависимостями - это внешние предупреждения :)
 - Могут быть ложные срабатывания
- Обновление компилятора может принести новые диагностики
 - А может - новое поведение неопределённое программы :)
 - Внутри компиляторов тоже есть свои “анализаторы”
- Для поддержания чистоты кода - `WARNINGS AS ERRORS`



Статический анализ кода

- Не требует компиляции программы
- Не надо запускать тестируемую программу
- Интеграция в CI
- Находит довольно интересные ошибки:
 - Одинарное '=' в if
 - Неправильные аргументы memset
 - Возможные переполнения переменных
 - Повтор условий в условных конструкциях
 - И много чего ещё :)
- Проверка кода на соответствие стандартам (MISRA, etc.)



```
template<typename Scalar> EIGEN_DEVICE_FUNC
inline bool isApprox(const Scalar& x, const Scalar& y,
    typename NumTraits<Scalar>::Real precision =
        NumTraits<Scalar>::dummy_precision())
```

```
template< .... >
void evalSolverSugarFunction(....)
{
    ....
    const Scalar psPrec = sqrt(test_precision<Scalar>());
    ....
    if (internal::isApprox(
        calc_realRoots[i], real_roots[j] ), psPrec)
    {
        found = true;
    }
    ....
}
```

```
void ExprParser::replaceBinaryOperands()
{
    char t1 = getOperandType(1);
    char t2 = getOperandType();
    popOperand();
    popOperand();
    if (t1 == t2)
        mOperands.push_back (t1);
    else if (t1 == 'f' || t2 == 'f')
        mOperands.push_back ('f');
    else
        std::logic_error
            ("failed to determine result operand type");
}
```

```
template<typename Scalar> EIGEN_DEVICE_FUNC
inline bool isApprox(const Scalar& x, const Scalar& y,
    typename NumTraits<Scalar>::Real precision =
        NumTraits<Scalar>::dummy_precision())
```

```
template< .... >
void evalSolverSugarFunction(....)
{
    ....
    const Scalar psPrec = sqrt(test_precision<Scalar>());
    ....
    if (internal::isApprox(
        calc_realRoots[i], real_roots[j] ), psPrec)
    {
        found = true;
    }
    ....
}
```

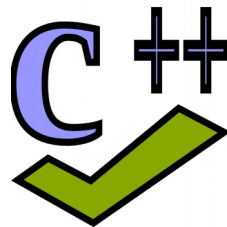
```
void ExprParser::replaceBinaryOperands()
{
    char t1 = getOperandType(1);
    char t2 = getOperandType();
    popOperand();
    popOperand();
    if (t1 == t2)
        mOperands.push_back (t1);
    else if (t1 == 'f' || t2 == 'f')
        mOperands.push_back ('f');
    else
        std::logic_error
            ("failed to determine result operand
            type");
}
```

Как готовить?

- Запускать **регулярно**
- Исправлять баги **постепенно**
 - “Задушить” пока что старые баги
 - Разметить вручную ложные срабатывания
- Встроить в CI
 - Настроить уведомление разработчиков о проблеме
 - Желательно, чтобы уведомление не игнорировалось :)
- По возможности использовать несколько анализаторов
 - Будьте аккуратны - это может быть довольно долго
 - И может быть больше ложных срабатываний => больше разметки



Статические анализаторы



- Встроенные в IDE
- cppcheck
- clang-tidy
 - Как писать свои проверки: https://www.youtube.com/watch?v=WHgXn_ufY90
- clazy
- Проприетарные средства (Coverity, PVS-Studio, etc)



Большой список смотрите здесь:

https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

Санитайзер - что за зверь?

Санитайзеры (sanitizers) - утилиты, которые идут в поставке нормальных компиляторов, которые могут найти ошибки в вашей программе **во время исполнения**.

- Требуют перекомпиляции программы
- Требуют запуска программы
- Требуют хорошего набора тестов
- Ведут к деградации производительности
 - Во время тестирования, разумеется :)

Санитайзеры - какие бывают

- **address** - находит ошибки работы с памятью (наиболее частые ошибки)
- **thread** - находит проблемы в многопоточном коде (гонки и deadlocks)
- **undefined** - находит неопределённое поведение в программе

Важно: не все санитайзеры друг с другом совместимы. Хотите проверить всеми - придётся пересобрать и запустить.

Как готовить санитайзеры?

- Запускать **регулярно**
- Обновлять - они тоже активно развиваются
- Встроить в CI
 - Настроить уведомление разработчиков о проблеме
 - Желательно, чтобы уведомление не игнорировалось :)
- Используйте все санитайзеры
 - Аккуратно - значительно замедляет процесс тестирования

Valgrind

- Фреймворк для построения анализаторов
- Это не только профилировщик
- Не требует перекомпиляции проекта
- <http://valgrind.org/>



Valgrind: утилиты (полезные нам)

- Memcheck - аналог Address sanitizer
- Helgrind, DRD - аналог Thread sanitizer
- Здесь может быть ваша собственная утилита

Представьте здесь какую-нибудь картинку, пожалуйста :)

Fuzzing

Фаззинг - техника тестирования путём подачи большого количества зачастую невалидных данных. А потом следим, как наше приложение обработает этот мусор.

- Помогает найти много ошибок
- Требуется много вычислительных ресурсов
- Эффективность **очень сильно** зависит от способа приготовления



Фаззинг: как ГОТОВИТЬ

- Соберите набор “валидных” данных
- Соберите словарь символов, из которых может состоять ввод
- Запускайте вместе с санитайзерами
 - Значительно замедляет скорость тестирования
 - Находит больше ошибок
- Держите запущенным длительное время



Фаззинг: утилиты

- American Fuzzy Lop (AFL)
 - <http://lcamtuf.coredump.cx/afl/>
- LibFuzzer
 - <https://llvm.org/docs/LibFuzzer.html>
- И несчётное количество остальных
 - <https://github.com/secfigo/Awesome-Fuzzing#tools>

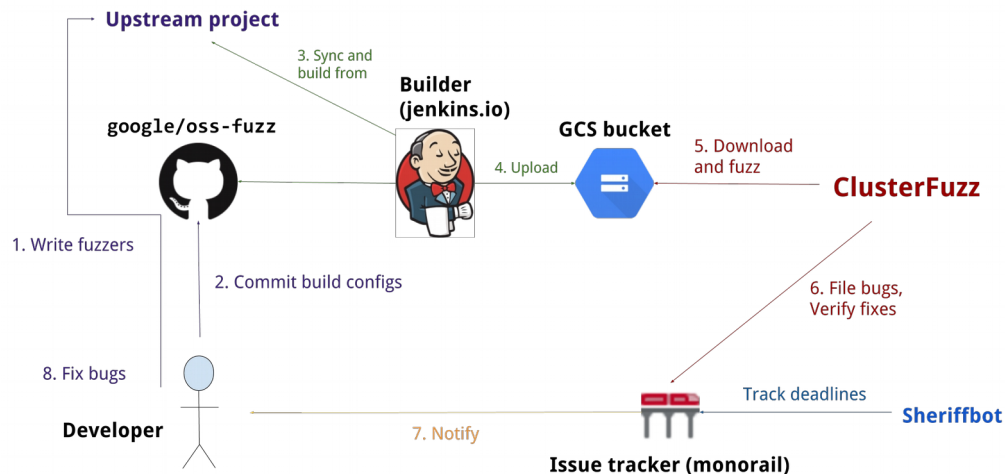
Пример

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t
size)
{
    std::string str((const char*)data, size);
    parse(str.c_str());
    return 0;
}
```

```
clang++ -fsanitize=fuzzer main.cpp
```

Oss-Fuzz

- Проект от Google
- Автоматизированный фаззинг на ресурсах Google
- Очень популярный проект для тестирования Open Source ПО
- <https://github.com/google/oss-fuzz>



ССЫЛКИ

- Kostya Serebryany “Beyond Sanitizers...”
<https://www.youtube.com/watch?v=qTkYDA0En6U>
- <https://github.com/secfigo/Awesome-Fuzzing>
- Дмитрий Вьюков “Fuzzing: The New Unit Testing”
<https://www.youtube.com/watch?v=FD30Qzd6ylk>



Итог

- Языки C/C++ при неправильном приготовлении опасны для здоровья :)
- Существуют различные подходы, которые позволяют “приручить” их, делая волосы мягкими и шелковистыми
- Знай те средства, что ты используешь
- Только в совокупности подходы приносят видимые плоды, позволяя отловить много ошибок до того, как их “оттестирует” пользователь :)

Спасибо за внимание!

И удачи в ловле багов в коде!

